DOCUMENT RESUME

ED 168 558 IR 007 093

AUTHOR Beard, M. H.; Barr, A. V.
TITLE The BASIC Instructional Program Student Manual.
INSTITUTION Stanford Univ., Calif. Inst. for Mathematical Studies in Social Science.
SPONS AGENCY Navy Personnel Research and Development Center, San Diego, Calif.
REPORT NO NPRDC-SR-77-2
PUB DATE Oct 76
NOTE 75p.; For related documents, see IR 007 092-096

EDRS PRICE MF01/PC03 Plus Postage.
DESCRIPTORS College Students; *Computer Based Laboratories; *Computer Science Education; Glossaries; Input Output; *Instructional Programs; Programing; *Programing Languages; Tutorial Programs
IDENTIFIERS *BASIC

ABSTRACT

This manual is the student's main source of information about the BASIC Instructional Program (BIP), a "hands-on laboratory" that teaches elementary programming in the BASIC language, and the BASIC language itself. The manual is organized as a reference document for students with no previous programming experience. Three major sections contain (1) an introduction to the course; (2) an explanation of general programming, discussions of programming concepts such as input and variables, and the specification of the BASIC statements used to implement these concepts, with the syntax and sample programs; and (3) a list and explanation of the commands that control the BIP system, some of which are identical to standard BASIC commands (e.g., RUN, LIST) while others give access to the unique features of BIP. A glossary is appended which lists all the specialized terms used in the manual, explains their use briefly, and gives references to the sections where detailed information can be found. (Author/CMV)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED168558

NPRDC Special Report 77-2

U S DEPARTMENT OF HEALTH, EDUCATION & WELFARE
NATIONAL INSTITUTE OF EDUCATION

THIS DOCUMENT HAS BEEN REPRODUCED EXACTLY AS RECEIVED FROM THE PERSON OR ORGANIZATION ORIGINATING IT. POINTS OF VIEW OR OPINIONS STATED DO NOT NECESSARILY REPRESENT OFFICIAL NATIONAL INSTITUTE OF EDUCATION POSITION OR POLICY

October 1976

THE BASIC INSTRUCTIONAL PROGRAM STUDENT MANUAL

M. H. Beard
A. V. Barr

Institute for Mathematical Studies in the Social Sciences
Stanford University
Palo Alto, California 94305

Reviewed by

J. D. Ford, Jr.

Approved by

James J. Regan
Technical Director

Prepared for

Navy Personnel Research and Development Center
San Diego, California

IR007093

FOREWORD

This report will serve as a handbook for students using the BASIC Instructional Program (BIP) which was developed at the Institute for Mathematical Studies in the Social Sciences, Stanford University. Initial research support for the program was provided by the Office of Naval Research and by the Defense Advanced Research Projects Agency. Current advanced development of BIP is being supported by the Navy Personnel Research and Development Center under Advanced Development Project ZPN07 (Education and Training Development), Subproject ZPN07.32 (Advanced Computer-Based System for Instructional Dialogues) under the mission sponsorship of the Chief of Naval Operations (OP-099). BIP incorporates several of the advanced techniques of computer-assisted instruction that are being tested and evaluated by ZPN07.32. Plans are currently being formulated for a BIP tryout at the U.S. Naval Academy. Dr. J. D. Fletcher served as technical monitor for this development.

J. J. CLARKIN
Commanding Officer

SUMMARY

The BASIC Instructional Program (BIP) is a "hands-on laboratory" that teaches elementary programming in the BASIC language. This manual is the student's main source of information about the BIP system and the BASIC language. The manual is organized as a reference document aimed at students with no previous programming experience. Section I introduces the student to the course itself. Section II begins with an explanation of programming in general. Discussions of programming concepts such as input and variables are followed by the specification of the BASIC statements used to implement them. The syntax and sample programs are used as illustrations. Section III lists and explains the commands that control the BIP system. Some are identical to standard BASIC commands (e.g., RUN, LIST), and others give access to the unique features of BIP. The glossary lists all the specialized terms used in the manual, explains their use briefly, and gives references to the sections where detailed information can be found.

TABLE OF CONTENTS

			Page
SECTION	I	INTRODUCTION TO BIP AND THE TERMINAL	1
	I.1	The BASIC Language and the BASIC Instructional Program (BIP)	1
	I.2	Using the Manual	1
	I.3	Signing On	2
	I.4	Talking to BIP	3
	I.5	A Sample Interaction with BIP	4
	I.6	Some Helpful Characters to Know	9
	I.7	Error Messages and Changing Your Program	11
SECTION	II	PROGRAMMING IN BASIC WITH BIP	13
	II.1	Programming	13
	II.2	Program Storage and Execution	15
	II.3	Line Numbers	16
	II.4	END	16
	II.5	Input/Output	16
	II.6	PRINT	17
	II.7	Data Types and Values	19
	II.8	Primaries	20
	II.9	BASIC Variables	22
	II.10	Assignment	22
	II.11	LET (Assignment)	24
	II.12	Expressions and Operators	25
	II.13	BASIC Operators	26
	II.14	INPUT	28
	II.15	READ . . DATA and REOPEN	29
	II.16	DIM	31
	II.17	Program Flow	33
	II.17.1	Loops	33
	II.17.2	Branch and Return	36
	II.18	GOTO	37
	II.19	Relational and Boolean Operators	38
	II.20	IF . . THEN	40
	II.21	FOR . . NEXT	41
	II.22	GOSUB . . BEGINSUB . . RETURN . . ENDSUB	44
	II.23	Functions, Arguments, and Returning Values	47
	II.23.1	Built-In Functions	47
	II.23.2	RND	48
	II.23.3	INT	48
	II.23.4	SQR	50
	II.23.5	LEN	50
	II.23.6	User-Defined Functions	51
	II.24	Other Useful statements	52
	II.24.1	STOP	52
	II.24.2	REM	52

TABLE OF CONTENTS (Continued)

SECTION III		BIP COMMANDS	53
	III.1	Curriculum Manipulation	53
	III.2	Program Manipulation	55
	III.3	File Storage and Access	59
	III.4	Dealing With the World	61
APPENDIX A		GLOSSARY	A-0
DISTRIBUTION LIST			

I.1 The BASIC Language and the BASIC Instructional Program (BIP)

This course is designed to help you learn some fundamental programming concepts through the BASIC language. BASIC is widely used; it is probably available on almost any computer system you are likely to encounter. BIP is an acronym for "BASIC Instructional Program," the program that runs this course. It is used only for this purpose and you will never hear of it in another context.

The version of BASIC used in this course is not identical with the many other versions you may find elsewhere. However, the fundamentals are the same, and the transition to another version of BASIC will be easy.

I.2 Using the Manual I.2

This manual is meant to be an easy and fast source of reference material. It will be most effective if you have it with you while you are working at the terminal. Try to become familiar with the manual, but do not try to memorize it. Keep it handy and refer to it often.

The first section of the manual introduces you to BIP and some of the keys on the terminal that you should know about. The main body of the manual is the second section, which explains fundamental programming concepts and structures and describes the language in which you will write your own programs (namely, BASIC). The third section lists and explains BIP's special commands. The fourth part is a glossary of all the terms used in the manual, with references to the appropriate sections for further information.

The manual is not intended to be a task-by-task guide to the course. It is a reference manual that contains a complete description of all the BASIC statements (the "sentences" of the language) and BIP commands. Especially when you first start programming, a reference manual contains a large amount of information that you are not ready to use. You must try to isolate exactly what you're looking for, and ignore information that doesn't seem to relate to your immediate problem. This is not easy, but it becomes easier with practice. The glossary is usually a good place to start.

Advice: Don't be afraid to make mistakes. A computer is a consistent machine, and you can frequently discover what works and what doesn't by trying different ways of doing something and watching the results carefully. The manual is full of sample programs that illustrate how BASIC works. Copy and RUN these programs whenever you like.

Whenever you want to use a computer, you always have to start by establishing communication with the machine somehow, letting the computer know who you are and what you want to do. To use BIP, you always start by "signing on" with your assigned BIP number. The details of the sign-on procedure will depend on where you are and what computer you are using to take this course, so you will have to find someone who knows these details.

In any case, these things are always true:

The terminal will say

WELCOME TO BIP!!

TYPE ? FOR HELP

*

and you have signed on. You will sign on in this way every time you work with BIP.

<cr> means the key marked "CR" or "RETURN" on the right side of the keyboard, and is an abbreviation for "carriage return."

The key marked "DEL" (on the right side of the keyboard) erases the last character you typed, like a backspace.

Once you have signed on, you will be "talking" to BIP. You must type <cr> to end each line you type. BIP reads and responds to your commands after you type <cr>. BIP types a * every time it is ready for you to type something.

It is not too soon to tell you about signing off. You must sign off before you leave the terminal. Do it by typing BYE <cr> to BIP. The terminal should print a short message ending with GOODBYE.

Please do not leave a terminal that has not said GOODBYE to you.

Occasionally, you will be the victim of a "system error" or a "system crash." These are unexpected, unpredictable, unavoidable events. You will know that one has occurred either because your terminal suddenly prints something like "SORRY, SYSTEM ERROR" or because your terminal stops printing anything at all. If you are near any other people using the same computer, you can ask them whether they are still getting any response; if they are, and you aren't, you should probably find the person who knows something about BIP.

BIP does not present lessons on programming. It does not ask questions and wait for you to type correct answers. It does present programming tasks that require you to write BASIC programs. By writing, running, testing, and fixing your own programs, you will learn a lot about programming. BIP will help you, not by knowing the correct answers (many different programs can produce the "right" result) but by identifying errors, giving you more information, and presenting tasks that build on the skills you have developed.

The pattern of the interaction between you and BIP generally goes like this:

a. You ask for a task, by typing TASK. BIP prints out the requirements for a program that it expects you to write and run.
b. You write the program, test it, fix it, test it, and complete it. You will make a number of errors along the way, many of which will cause BIP to print an error message, telling you that it can't understand what you typed, or can't do what you told it to do.
c. Having written the program required by the task statement, you type DONE, and BIP looks at your program to see that it works as it should. BIP then completes the task by giving you the "post-task interview." In some cases, the current task will be extended with some additional requirements.

Within that pattern, many additional things may happen. You may be confused, either by the statement of the problem, by the error messages printed by BIP, or by your program doing something you do not expect when you run it. There are specific ways to deal with each kind of confusion.

a. If the task is not clear, you may request a HINT or use DEF. To get a better idea of the output your program should produce, you may run a DEMO.
b. If the meaning of the error messages is not clear, you may type a question mark (?) just after the message is printed. BIP will give you some brief information about that error. You should also look in the glossary for more information, starting with the first word or phrase that you don't understand. Use the cross-references to find the information you need.
c. If your own program does not behave as you want it to, you should first LIST it to see exactly what it says. You should also use one of the debugging options—TRACE or FLOW—which will show you, in detail, your program in action.

(continued)

If you are already confused just by reading this, go to the glossary at the end of the manual and start looking up the words printed here in upper case, like THIS. Do not try to memorize anything. Do try to get an idea of how you can find information in the manual.

I.5 A Sample Interaction with DIP

This dialogue should give you an idea of what it's like to work with DIP and illustrate how you can learn from your errors. The student's typing is underlined; everything else is upper case is printed by DIP. (You will never have to type underlines yourself when you use DIP.) Comments and explanations appear at the right in lower case. Remember that each line of student's typing ends with a <cr>.

Start by signing on. You hit the space bar, and the dialogue begins. (This is the Stanford version. If you are somewhere else, yours will differ.)

```
@DI

PLEASE TYPE YOUR USERNAME AND PASSWORD.
1234 CHARLES BAGLE
==============
  JOB 11 ON TTY332,  THU DEC 10 73 9:31AM-PST

                WELCOME TO DIP !!

TYPE "?" FOR HELP
                                  A reminder that information is always
                                  available.

@? CHARLES!

*FIND                             You ask for something to do.
=====
BASIC TASK: STRING
FIND THE -SECB- CONTAINED IN SCRATCH (CSAIA) USING PATTERNS PREVIOUS, IF ANY.

WRITE A PROGRAM THAT PRINTS THE STRING "GLORIA".  START WITH THE
GLOSSARY TO FIND OUT WHAT A STRING IS.

                                  DIP describes a program for you to write.

                                                (continued)
```

[illegible]

[illegible]

[illegible]

[illegible], and one of the old programs that's been there a while knows a procedure name.

[illegible]

You're not sure how to start, so you ask for a hint.

[illegible]
[illegible] [illegible] [illegible] [

```
*MORE
 ----
YOU HAVE NOT RUN THE PROGRAM.
```

But you didn't -RUN- the program. You haven't actually seen it print "SCHOOL" yet.

```
*RUN
 ---
```

Try again.

```
EXECUTION OF YOUR PROGRAM

SCHOOL

EXECUTION COMPLETED AT LINE 20
```

Now it looks fine. Your program ran without any errors, and it printed the string.

```
*WHAT
 ----
```

You want to make sure that you met the requirements of the task, so you ask for its text again.

```
YOU ARE IN TASK STRINGY.
DO YOU WANT THE TEXT PRINTED OUT?  TYPE Y OR N.
        >Y
         -
```

Yes, please.

```
USE THE -SCR- COMMAND TO SCRATCH (ERASE) YOUR PREVIOUS PROGRAM, IF ANY.

WRITE A PROGRAM THAT PRINTS THE STRING "SCHOOL".  START WITH THE
GLOSSARY TO FIND OUT WHAT A STRING IS.
```

As you thought. You can always use the -WHAT- command to review the task.

```
*MORE
 ----
```

You're ready to go on.

```
EXECUTION OF THE MODEL

EXECUTION OF YOUR PROGRAM

LOOKS OK!!
```

BIP compares your program with the model, and sees that yours is fine. Now it asks you to evaluate yourself.

```
POST TASK INTERVIEW

HERE IS A TYPICAL SOLUTION TO THIS TASK:
        10 PRINT "SCHOOL"
        99 END
```

(continued)

(I.5) (I.5)

DO YOU UNDERSTAND THE SOLUTION? (Y/N): Y
-
You understand why the model works.

THINK ABOUT THE SKILLS USED IN THIS TASK. FOR EACH SKILL,
TYPE Y IF YOU HAVE HAD ENOUGH WORK WITH THAT SKILL.
TYPE N IF YOU THINK YOU NEED MORE WORK ON IT.

PRINT STRING LITERAL : N
-
You think you'd like to do more with strings and quotation marks. BIP will remember that fact; you can expect more strings later.

TASK STRINGY COMPLETED.

The end of this task.

*TASK
You ask for another.

BEGIN TASK: PLUSFOUR
THIS PROGRAM SHOULD ASSIGN THE VALUE 6 TO THE NUMERIC VAPIABLE N.
THEN PRINT THE SUM OF N AND 4.

You see some unfamiliar terms, and realize that you have to spend at least a little time with the manual.

*BYE
You also realize that you don't have any more time, so you sign off.

SIGNOFF: THURSDAY, DECEMBER 18, 1975 9:41:38AM-PST

YOU HAVE COMPLETED 1 TASK TODAY:
STRINGY

TOTAL TIME TO DATE: .800 HOURS
TIME ON TODAY: .067 HOURS
TOTAL SESSIONS: 2
TOTAL TASKS COMPLETED: 3

DR. MEMORY SAYS: "CONSIDER YOURSELF AT HOME."

COPYRIGHT (C) 1973 BY THE LELAND STANFORD JUNIOR UNIVERSITY
GOODBYE, CHARLES.

And that's all.

I.6 Some Helpful Characters to Know

The key marked "CTRL" on the left side of the keyboard works like a shift key. You must hold it down while you type a letter. For example, to type CTRL-X you have to hold the "CTRL" key down while you type X. CTRL is pronounced "control," and letters typed with the "CTRL" key are called "control characters." CTRL is abbreviated with this mark: ^
so ^X means the same thing as CTRL-X.

A control character is a special type of command. You may use the following control characters at any time:

^A (or the delete key, marked DEL or DELETE) erases the last character you typed.

^X "erases" the line you are typing, as far as the computer is concerned. It is useful if you discover a mistake you made at the beginning of the line. After typing ^X, you should start typing the line again from the beginning.

Note: the following control characters have to do with "editing" the line you are typing. Editing is much easier to demonstrate than to explain, so as soon as you get a chance, try out the example shown below under ^N.

^W erases the word you are typing, but not the whole line. After typing ^W, start that last word again. (A word is everything back to the last space, but not including the space.)

^R retypes the current line. ^R is useful if you've erased words or characters and you don't know what the line really looks like.

^E retypes the last line you typed and adds it to the current line. This is so you won't have to type an incorrect line all over again. Use the -EDIT- command to make any line of your program as if it were the last line you typed in. For example, EDIT 70 will "bring back" line 70. BIP will print the line so you can see exactly what it is, then allow you to change it as you please.

^N retypes the next word of the last line you typed. (A word is everything back to the last space, but not including the space.)

(continued)

Example: suppose you just typed:
10 PRINT "THE FOX JUMPED OVER THE LAZY DOG"
and you wanted to change it.
Type ^N three times to get
  10 PRINT "THE
then type QUICK BROWN (followed by a space) to get
  10 PRINT "THE QUICK BROWN
and then type ^E to get the end of the line tacked on:
  10 PRINT "THE QUICK BROWN FOX JUMPED OVER THE LAZY DOG"
Try it on your terminal. Also see the example in Section I.7.

Before you type ^E or ^N on a line you have already started, put a space at the end of the line so the words don't run together.

^S skips the next word of the last line you typed. Same as ^N followed by ^W.

** Memorize these three control characters now. You will need them:

^Z signs you off. Emergency escape if your terminal "hangs up".

^G interrupts the running (execution) of your program. You will occasionally write a program that never reaches its end, typing or computing interminably. Typing ^G may be the only way to force it to a halt. You may want to "abort" your program with ^G at other times, too--any time you don't want it to run all the way to the END.

HOLD stops the screen so that you can read everything before it disappears off the top.
There may be a key marked "HOLD" on your terminal. If not, use ctrl-shift-O (push the shift key and the ctrl key at the same time, and type the letter O while they are down). You don't need to hold the key(s) down to freeze the screen. Just hit HOLD once and the screen will stop within a second or so. When you want to start the screen moving, hit HOLD again. Any other character will also start the screen moving after you stop it, but that character will also print on the screen. Ignore it.

I.7 Error Messages and Changing Your Program I.7

"Errors" were mentioned earlier. In the context of this course, an error is something that BIP knows it cannot handle correctly. For example, if you type something like "RASK" when you meant to type "TASK", BIP will give the error message ILLEGAL BIP COMMAND because it can't do anything with the word "RASK". There are three different kinds of errors that BIP detects and tells you about:

a. "Syntax errors" are detected immediately after you complete your line. There are rules that you must follow when you give a BIP command (like the one above) or type a BASIC statement. BIP recognizes violations of those rules and complains immediately. (An error you may make frequently is to misspell a word, as in the example.)

b. "Error Doctor errors" are detected when you tell BIP to RUN your program. A program is a list of instructions for the computer to follow; if your program is missing some essential things, the computer can't follow the instructions. BIP recognizes the absence of these essential things, and tells you what's missing.

c. "Execution errors" are detected as your program is running. If your BASIC program turns out to be impossible to follow at some point, BIP will try to tell you what the problem is.

It is a good idea to LIST your program before you make any changes. You must make some changes if BIP prints an error message, or if the program does not produce the results you want. To make a change, retype correctly the line with the error. Suppose you had the line

```
50 PRINT X
```

and you decided (or BIP forced you) to change it to

```
50 PRINT Y
```

instead. You could either retype the whole line, substituting Y for X, or you could -EDIT- the line (see I.6) like this:

You type EDIT 50, and BIP prints the line as it is now:
50 PRINT X
It also prints the line number for you, so now you type ^N to get
50 PRINT
Then you type the Y at the end, followed by <cr>, and you get
50 PRINT Y

BIP will always tell you what the line was before the change, as a warning in case you didn't really want to change that line. (If this is the case, you must change it back again.)

If you want to delete a line completely, type the line number and the "CR" key. Then LIST the program to be sure you have what you want.

SECTION II. PROGRAMMING IN BASIC WITH BIP

This is the main body of the manual. It is organized by complexity of concepts--the most fundamental first, the more advanced later. Since programming concepts frequently overlap, however, you will have to bounce forward and back to find the information you need in a particular situation.

Do not try to memorize the information, especially the first time you read this section. You may not even want to read this entire section of the manual at one time. Sub-sections that should be read together, if you choose to read chunks at a time, are:

II.1 - II.4	Some fundamentals of programming in BASIC.
II.5 - II.11	Input, putput, assignment, and variables.
II.12 - II.13	Expressions.
II.14 - II.16	INPUT and READ statements.
II.17 - II.20	Sequence and control of execution.
II.21 - II.23	FOR, GOSUB, and functions.

Read II.24 the first time you see STOP or REM in the model solution.

II.1 Programming II.1

A computer is not smart. It can only do what it is instructed to do, and every tiny step must be communicated in a form that the computer can understand. A program is a list of instructions to a computer.

Writing a program involves three big stages:

a. Specify in complete detail what the program is supposed to do.
b. Translate your statement of the problem into a language the computer understands.
c. Check the program to be sure that it does everything you want it to do.

The difficulty of each stage relative to the others may [illegible] but none of the three can ever be ignored just because the programmer thinks "it's too easy." In particular, you must not neglect the first stage, the detailed description of the problem. It is often useful to write out in English exactly what you want the program to do, and in what order. You should list the steps you would have to follow to solve the problem by yourself; if you cannot do this, you will not be able to use a computer to solve the problem. For example, you can ask a friend to give you two numbers, and you can tell him the result of multiplying those numbers together. If you think about it, you can see that there are a number of steps involved:

Ask for the first number
Hear it and remember it
Ask for the second number

(continued)

Hear it and remember it
Multiply and remember the result
Tell your friend the result

The more specific you are in describing each step of the problem, the easier it will be to complete the second stage, where you translate your English into a programming language. A computer cannot understand English, nor can it guess at your meaning if you give it an instruction that is only close to what you meant. The rules governing the syntax, or grammar, of programming languages are rigid, and you must use the correct words, the correct punctuation, etc. Just remember that your English list of steps, although essential, is not yet a computer program; you must translate each step into a series of symbolic instructions in exactly the form that the computer, through a programming language, can accept. This becomes much easier with practice, just as in any other foreign language.

The third stage in writing a program, where you check everything to be sure it all works as you want it to, is as necessary as the other two. The computer will follow exactly the instructions you give it. If these instructions do not say precisely what you meant, the program will not quite do what you want. Because programs must be so precise, it is easy to overlook small but important details, and very few programs run "correctly" the first time. No computer will make up for your negligence, so you must check the results of your program at least as carefully as you thought out the problem in the first place. This process, called "debugging," is tedious but necessary. If a program doesn't work, it's usually the programmer's fault, not the computer's.

In many programming languages, you first write your list of instructions, and then tell the computer to follow all the instructions in the list. Your list is sometimes called a "stored program" because the computer must store the instructions up until the time that you tell it to begin executing them. Execution is called "running" the program.

Whether the purpose of the program is to perform complicated calculations or to play a simulated card game, it must have some information on which to operate. This information is called data, and much of the data required by a program can be stored in the program itself. The alternative to storing the data in the program is to have the user (the person who runs the program) supply some data when the program stops and asks for it.

For example, a program whose purpose is to print a 10 by 10 multiplication table should have all its information stored within it. It is not necessary to request information when the program is actually executed--the user simply tells the computer to run that particular program. In contrast, consider a program that plays a game with the user. Such a program needs to get information as it runs, since the progress of a game cannot be planned in advance. The program must stop and ask the user for information--what move he wants to make, for example. This second kind of program is called "interactive" because it requires the programmer to plan for interaction with the user of the program as it runs.

In either type of program, the data that the program deals with must be kept in the computer in a way that makes it accessible to the program. This is done by the use of variables of different data types, which are discussed specifically in Sections II.7 through II.9.

A word about "the user": Programmers usually write programs for other people to use. Whether the program calculates payroll checks or plays a card game, it will be used by someone other than the person who wrote and debugged it. As you write your own programs, remember this hypothetical person called "the user." Try to make your programs understandable and complete enough that a friend of yours could sit down and run them without any trouble.

It's also a good idea to include "remarks" inside your program, with the -REM- statement. A remark (also called a "comment") is very simple: it's just a note to yourself that explains something about the program without affecting the way the program runs at all. You will be surprised to see how soon you can forget what an "old" program (a week old, for example) is supposed to do. REMarks that are saved as part of the program itself are handy notes to remind you.

(continued)

It is not hard to write a program that does the same thing over and over, never stopping. To force a program like this to a halt, hold down the "CTRL" key while you type the letter C. A program that never stops is in an "endless loop" which you must stop or "interrupt." The -GOTO- section (II.18) has an example of this kind of loop.

II.3 Line Numbers II.3

Almost all implementations of BASIC require you to number each line of your program. Each line, or statement, is an instruction to BASIC, telling it to do some specific thing. When you run a BASIC program, BASIC finds and obeys the instruction with the lowest line number, then the one with the next higher number, etc. You need not type in your statements in order, because BASIC can sort them out by line number, but you must number them in the order you want BASIC to follow. A general practice is to use multiples of 10 as your line numbers so that you have plenty of numbers available if you want to insert something between two already existing lines. BIP allows you to have up to 500 lines in a single program, but most programs will be much shorter.

II.4 END II.4

Use: To tell the computer when it has finished executing your program.

Example:

```
99 END
```

Remarks:

Every BASIC program must have an END statement. The END statement must have the highest line number in the program.

See STOP (II.24.1).

II.5 Input/Output II.5

This term refers to the problem of communicating with the computer—how you tell it to do something for you, and how you make it deliver the results in a way you can understand. Most people communicate with computers through programs, so the subjects of input and output really deal with providing information to your program that makes it provide meaningful information to you.

(continued)

.nput is information that goes into the program. It can be stored (as part of the program itself) when the program is written, or given by the user when the program is run.

Output is the visible result of a program's execution. It is frequently in the form of information printed on the user's terminal, or it may be transmitted to a lineprinting device, or to a magnetic tape, etc. In the case of interactive programs, it is important for the programmer to remember that the output his program prints will be read by someone else, and must be reasonably understandable. A dialogue between a person and a computer is pointless if neither understands what the other says.

II.6 PRINT

Use: To get your program to type something on the terminal.

Examples:

```
40 PRINT 4

40 PRINT Y + 10

40 PRINT A$

40 PRINT "DOG"

40 PRINT 10 < 15

40 PRINT "THE VALUE OF X IS "; X; " AND X SQUARED IS "; X^2

40 PRINT                (prints a blank line.)
```

Remarks:

Use the PRINT statement whenever you want to have your program type something. Anything surrounded by quote marks is taken literally. Anything without quote marks is "evaluated"--BASIC figures out what its value is, and PRINT prints that value.

The statement

```
40 PRINT "X"
```

prints just the letter X, because of the quote marks. The statement

```
40 PRINT X
```

makes BASIC look up the value of the variable X, then print that number. There are no quote marks, so BASIC has to evaluate X. (Read about values, variables, and evaluation in the next few sections.)

(continued)

Boolean values can be printed too. The statement

```
40 PRINT 10 > 9
```

prints TRUE on the terminal, because 10 is greater than 9.

```
40 PRINT 10 = 100/2
```

prints FALSE, because 10 is not equal to 100 divided by 2.

"Fancy" PRINT statements:

Using a semicolon between two expressions allows you to print more than one expression on a single line. You may combine different types of expressions in a PRINT statement. The semicolon allows you to PRINT both literals and variables in one statement, which can make your program's output look good. For example, you could use two PRINT statements like this:

```
40 PRINT "X IS"
50 PRINT X
```

which would tell the user of the program the value of the variable X, but would take two lines of output to do it. A nicer way to do it would be like this:

```
40 PRINT "X IS ";X
```

which would give the same information, but all on one line.

A more complicated example: Assume that the variable X has the value 4, and the variable Y has the value 5. The statement

```
40 PRINT "THE SUM OF YOUR NUMBERS IS ";X+Y
```

will cause BASIC to print

```
THE SUM OF YOUR NUMBERS IS 9
```

The statement

```
40 PRINT "X + Y = 15 IS ";X+Y=15
```

will cause BASIC to print

```
X + Y = 15 IS FALSE
```

Remember to use spaces inside your quotation marks where you need them. Some implementations of BASIC insert a space for every semicolon, but DIP's BASIC does not.

See Variables (II.8 - II.11) and Expressions (II.12, II.13, II.19).

11.7 Data Types and Values

Most programming languages operate on three different types of information: numeric, string, and Boolean. Many languages do not allow the programmer to combine different kinds of information in a single expression, and it is essential that you understand the differences.

Numeric information is easy to understand. A number or a numeric expression is a thing that you can add, or find the square root of.

A string is a series of characters in a particular order. (A character is something a typewriter can generate, including letters, numerals, punctuation, and spaces.) You cannot add or multiply strings as you can numbers, although most languages allow you to perform some operations on strings. In the course you are taking, your name is stored in the computer as a string, which is why the terminal can type your last name for you when you sign on. A string expression is a thing that has this kind of value, as opposed to a numeric value.

Boolean information is understood by the computer to be either true or false. In most programming languages, you can tell the computer to do one thing if something is true, and another thing if it is false. The value of a Boolean expression is always either true or false. (The word "Boolean" comes from the name of a mathematician named Boole.)

A word about the size of numbers and the length of strings in BIP: Although you can use very large numbers (20 digits, for example), BIP is only accurate to 10 places, so very large numbers involve very large errors. Your strings can be quite long (100 characters, for example), but you only have room for about 60 characters on a line. So you should keep your numbers to a size of 10 digits or less, and your strings to 60 characters or less. Boolean information has no size to speak of, so enough has been said.

While your program is executed, the computer must be able to know, or to find, the value of all the pieces of information in it. As described in the previous section, these values may be numeric, string, or Boolean.

The information that your program deals with can be extremely simple, or extremely complex, or anything in between. A primary is the simplest kind of information that you can talk about, because the computer must go through or down one step to find the value. Numeric and string primaries exist in almost all programming languages, either as literals (also called constants) or as variables, requiring assignment of values.

Literals are very straightforward. What you see is what you get; a literal is taken literally. A numeric literal is what you immediately recognize as a number: for example, 7 or -6.8. A string literal is enclosed in quotation marks and is something you immediately recognize as a sequence of characters: for example, "BOB", or "Hello!". The only slightly tricky thing about string literals is that the characters may be numerals, but the value of the string is still a string, not a number: for example, "5" cannot be added or multiplied. "5" is just something that can be printed, like "X" or "BOB".

The other kind of primary is the variable. Variables are used as names for values or as "boxes" to hold values. The value of a variable is either a number or a string, depending on what was assigned to the variable.

There are two kinds of variables. A <u>simple</u> variable is a "box" that holds one value, either one string or one number. A <u>subscripted</u> variable (often called an "array variable") can hold many values, in order, all under the same name.

Simple variables are like the simple types below. The first one is a numeric variable, because the value in the box is a number. The second is a string variable, because the value in the box is a string.

```
 A  | 15 |          A$  | "HELLO" |
```

In this example, the value of the variable A is 15, and the value of the variable A$ is "HELLO". (BASIC string variables always have that dollar sign. The variable A$ is pronounced "A string" or "A dollar".)

(continued)

Subscripted variables are like the multiple boxes below. Each box has only one name, but the only essential thing is which box. Each box can hold a value of its own.

In this example, the value of B(1) is 6, B(2) is 8, and B(3) is 5. The variable B is being used to hold a list (or "array") of numbers. The string variable SM is being used to hold a list of pairs of characters. The value of SM(1) is "HE", SM(2) is "SH", and SM(3) is "AD". B(1) is pronounced "B sub 1" and SM(3) is pronounced "S M sub 3."

Each of the elements of a subscripted variable can be treated as a separate variable. The value can be changed by an assignment statement, it can be compared to another value, it can be printed, etc. Subscripted variables can have as many elements as "space" we give them. See 11.10 for more information about this.

The important thing to remember about both kinds of variables is that they do not receive and remember calculations. In the case of literals, the value is simply the literal itself--nothing is hidden. In the case of a variable, the computer can find the value immediately by looking in the "box" named by the variable, where the value is stored.

See BASIC Variables and Assignments (10.8 + 10.10).

II.9 BASIC Variables II.9

Use: To name locations (or "boxes") where values are stored.

Examples:

```
X
X2
A$
```

Remarks:

A numeric variable names a "box" whose contents must have some numeric value--for example, -6 or 2.5--that can be changed by arithmetic operations like addition or division. A numeric variable must be a single letter or a single letter and a single digit. In the above examples, X and X2 are numeric variables.

A string variable names a "box" whose contents must have some "string" value--for example, HORSE--that can be changed by the string operation called "concatenation". A string variable must be a letter followed by the $ character. In the above examples, A$ is a string variable.

See Primaries (II.8) and Assignment (II.10 - II.11).

II.10 Assignment II.10

All programming languages make extensive use of variables, the "boxes" used to hold values. A program that deals only with literals cannot be used in any kind of general way, since nothing within the program can ever change. For example, a program that adds 2 + 4 has limited use, but a program that uses variables to hold the values of two numbers, then adds them, is obviously more useful, since that program can add any two numbers.

The mechanism by which variables are given values is called assignment. The simplest form of assignment is this:

<variable> = <literal>

For example:

```
X = 5
```

After this assignment is done, the variable X "has the value" 5. Any reference to X (like printing it, or adding 1 to it) is actually a reference to the "box" whose name is X, the box that now has 5 in it. The value of X can be changed by another assignment, after which every reference to X will be taken as a reference to that new value.

(continued)

(II.10)

The value assigned to a variable can be given as an expression combining two or more values. Thus the value of X could be assigned as

```
X = 5*4
```

or, assuming that the variable Y had already been assigned a value of its own,

```
X = Y + 1
```

When the computer executes an assignment statement, it follows these steps:

a. Evaluate the expression on the right side of the "=" sign.
b. Put that value into the "box" named by the variable on the left side of the "=" sign.

Thus, the assignment `X = Y + 1` means, "Find the value of Y, and add 1 to it. Then assign that result as the value of X." Note that the value of Y is not changed by this assignment. Only the variable on the left side of the = sign gets a new value. (Do not confuse your right and left hands, or your variables will seem to have strange values.)

The assignment `X = X + 1` means, "Find the current value of X. Add 1 to it, and assign that new value to X." If X had the value 5 before the execution of the assignment statement, then it would have the value 6 after the assignment.

The contents of the variable on the left side of the "=" sign are always replaced by the value of the expression on the right side. The old value of the variable (whatever value it had before the assignment statement) is lost.

Use: To give a value to a variable.

(Note: In BIP's BASIC, you may use either the "=" sign or the "_" sign in assignment statements. The "_" will print as an underline or as a left-arrow on your terminal, and you type it by holding the SHIFT key and typing the letter O.)

Examples:

```
10 LET X = 5

10 B$ _ "HELLO"

10 A2 = A2 + 1

10 X$ (1) = "RAINDROP"
```

(The word LET is optional.)

Remarks:

BASIC variables are assigned values as explained above in II.10. Note that the "=" sign does not indicate equality in this context; instead, in assignment statements, "=" and "_" mean something more like "becomes" or "has the value of."

The assignment statement in BASIC is called the LET statement, to remind you that

```
LET X _ 5      and      X = 5
```

both mean "Let X have the value 5."

Remember that right and left are different, and that

```
M$ = N$
```

means "find the value of N$. Assign that value to M$." This LET statement will not change the value of N$.

A statement like

```
100 = X      or      "DOG" = M$
```

will cause a syntax error from BIP, because you can't assign a value to 100 or to "DOG". If you want the value of X to be 100, you should say `X = 100` and if you want the value of M$ to be "DOG", you should say `M$ = "DOG"`.

See Data types, Primaries, and BASIC Variables (II.7 - II.9). Also see DIM (II.16).

II.12 Expressions and Operators

A primary can be either a variable or a literal. In either case, the computer must go through at most one step to determine the value of a primary. An operator is a symbol that tells the computer to combine or compare two primaries in some way.

Using these definitions, an expression can be defined as either a primary

Examples: "CAT"
B

or a primary followed by an operator, followed by an expression.

Examples:
```
X + 1
W$ & "SONG"
(6+4) * 9
((6 + Y2) - (A + B)) / X
"DOG" & (F$ & W$)
R$ (1, 3) & R$
(A >= B) OR (D$ = "DOG")
```

Using the term "expression" in its own definition means that an expression can be almost infinitely complex. Programming languages follow a process of evaluating each part of the expression, and then putting it all together to find the value of the expression as a whole. (Think of how you determine the meaning of a complicated phrase like "the sister of the father of my brother's sister's son's mother." A computer determines the meaning, or value, of each part of an expression in a similar way.)

More complicated expressions are evaluated from left to right and, when necessary, according to the following rules:

a. Expressions within the innermost parentheses are evaluated first.
b. Exponentiation (^) is done before any other operations.
c. Multiplication (*) and division (/) are done next.
d. Addition (+) and subtraction (-) are next.

This means that you may need to use parentheses to make the computer evaluate an expression correctly. In addition, you should always use spaces and parentheses to make your expressions easy for you to read. Extra spaces or extra pairs of parentheses will not cause errors.

Some examples:

```
5 + 3 / 2 ^ 2   is evaluated as   5 + (3 / (2^2)) = 5.75
((5+3) /2) ^2   is evaluated as   (8/2) ^2 = 16
```

(continued)

One essential thing to remember about using operators in programs is that you must be explicit. Although normal algebraic notation like

A + 2B

is clear to you and your algebra book, it is not clear to the computer. Any time you want the program to perform multiplication, you must say so, usually with the multiplication operator "*" . The equivalent of the above algebraic expression is

A + 2*B

You will also quickly notice that your terminal cannot type exponents up above the base. Exponentiation is always indicated on the same line, usually with the "^" operator. (On some terminals, there is a key with an arrow that points upward. On others, you get the "up arrow" with shift-N.) Thus, to get 17 squared, you must use

17 ^ 2.

(Remember, spaces are optional. 17^2 is also 17 squared.)

See Operators and Operations (II.13, II.19).

II.13 BASIC Operators II.13

A BASIC operator can be one of many different things. The arithmetic or numeric operators are

^	exponentiation
*	multiplication
/	division
+	addition
-	subtraction

The arithmetic operators work in BASIC just as they do in other programming languages, as explained in II.12.

The BASIC string operators are

&	concatenation
(X, Y)	substrings

Concatenation is used to join together two strings. For example, suppose the value of the string variable A$ is "HELLO ". (Notice the space after the "O".) And suppose the value of the variable B$ is assigned this way:

B$ = A$ & "THERE."

The concatenation of A$ and "THERE." would make the value of B$

"HELLO THERE."

(continued)

(II.13)

Some advice about concatenating strings: If you are putting words together (as in the HELLO THERE example), don't forget about the space between the words. If you concatenate "CAR" and "WASH" this way

```
"CAR" & "WASH"
```

the result is "CARWASH", which may be just what you want. If you say

```
"WELCOME" & "HOME"
```

you get "WELCOMEHOME", which is probably not what you want. You can say either

```
"WELCOME " & "HOME"              (space after "WELCOME")
```

or

```
"WELCOME" & " HOME"              (space before "HOME")
```

or

```
"WELCOME" & " " & "HOME"         (space quoted by itself)
```

all of which result in "WELCOME HOME". This concatenation

```
"WELCOME"&" "&"HOME"
```

produces the same "WELCOME HOME" result, because the space is inside the quote marks as in the other examples. A space inside quote marks is just like any other character and becomes part of the resulting string just as any letter would. Using spaces to separate different parts of your expression makes your lines easier to read, but has no effect on how the expression is evaluated.

A substring is a part of a string. In the example above, X and Y refer to the "start" and "stop" characters in the string. For example, "PURPLE" (1, 3) means the first through the third characters in the word PURPLE. The value of "PURPLE" (1, 3) is "PUR". "PURPLE" (4, 5) is "PL". The numbers can be variables, so if the value of X was 3 and the value of Y was 5, then "PURPLE" (X, Y) would be "RPL". The string can be a variable too, so if the value of H$ was "PHANTOM", then H$ (X, Y) would be the same as "PHANTOM" (3, 5) and the value would be "ANT".

This substring "BEAN" (5,5)
would be the fifth character in the string "BEAN" if there were five characters to begin with. If you specify a nonexistent substring like this one, the result is nothing. (See II.14 for an explanation of the "null string.")

This substring "BEAN" (3,2)
would be the third through the second character in the string "BEAN", if BIP could count characters backwards, but it can't. An "impossible substring" like this one will cause an execution error when BIP tries to evaluate it.

BASIC cannot evaluate an expression that contains different types of values. For example, this expression has no meaning

```
9 + "NINE"
```

because 9 is a numeric primary and "NINE" is a string primary.

See Data Types and Primaries (II.7 - II.8), Variables and Assignment. (II.9 - II.11), and Boolean Expressions (II.19)

Use: To allow the user of the program to give a value to a variable.

Examples:

```
30 INPUT N          (for a number)

30 INPUT P$         (for a string)

30 INPUT X, B$      (for multiple input)
```

Remarks:

When the INPUT statement is executed, BASIC types a colon (:) and waits for the user to type something, ending with the RETURN key. Whatever the user types becomes the value of the variable in the INPUT statement.

The only limitation in the use of INPUT involves numeric variables and is imposed when someone runs the program. if a numeric variable is specified in the program, the user must type a single number, not a string or any kind of expression. Numbers like 1492 or 6.25 will be accepted, but an expression like 3*4 will not. BIP prints an error message and lets the user try again.

This program doubles any number the user types:

```
10 PRINT "TYPE A NUMBER AND I'LL DOUBLE IT FOR YOU"
20 INPUT Y
30 Y = Y*2
40 PRINT Y
99 END
```

This program does something simple with a string typed by the user:

```
10 PRINT "TYPE A FEW WORDS AND I'LL REPEAT THEM"
20 INPUT W$
30 PRINT W$
99 END
```

Note: when typing a string in response to an INPUT, the user should not type quotation marks. Also, for strings, if the user types only the "CR" key, the string variable is assigned the value "". This is called the NULL string. The null string is analogous to the number 0 (zero). It is a known value, something that has meaning: it means the string version of nothing just as zero means the numeric version of nothing. Do not confuse the null string with the character " " which is a space.

(continued)

(II.14)

One INPUT statement may be used to allow the user to give values to more than one variable. For example, this program accepts two numbers and adds them.

```
10 PRINT "TYPE TWO NUMBERS, ONE AT A TIME."
20 INPUT X, Y
30 PRINT "THE SUM IS "; X+Y
99 END
```

You may specify as many variables in a "multiple input" statement as you like, always separated by a comma. When BIP's BASIC executes this statement, it prints a colon for each value to be typed by the user. Other implementations of BASIC work in a different way.

See Input / Output (II.5) and Variables (II.9).

II.15 READ . . DATA and REOPEN

II.15

Use: To assign stored values to variables.

Examples:

```
10 READ X
50 DATA 200
```

```
10 READ P
20 READ Q
30 READ R
200 DATA 5, 20, 50
```

```
30 READ A, B$
80 DATA 60, "DOG"
```

```
60 REOPEN
```

Remarks:

Using READ and DATA combinations allows you to store values in the program and assign those values to variables at appropriate times. The statement

```
READ X
```

causes BASIC to take a value from the DATA statement and assign that value to the variable X. For every execution of a READ statement, there must be a corresponding DATA value.

(continued)

As shown in the second example above, a DATA statement may contain more than one value. BASIC keeps track of the DATA values, and after a READ is executed, BASIC moves a pointer to the next value in the DATA statement. In that second example, the variable P would get the value 5, Q would get 20, and R would get 50.

The third example shows a multiple READ statement. Execution of a multiple READ assigns values to both variables, just as if one READ immediately followed the other. In the example, execution of line 30 would result in the assignment of 60 to the variable A and the assignment of "DOG" to the variable B$. Use multiple READ statements whenever you wish to assign values to more than one variable, all at the same time.

If a READ statement is executed, and all the DATA values have been "used," an execution error message will be printed (since no value remains to be assigned). To avoid this error, use a "dummy" value at the end of the DATA list and stop READing after that value has been used. In this program, -1 is used as the "dummy" that marks the end of the list of DATA values.

```
10 PRINT "THIS PROGRAM PRINTS SQUARES"
20 READ Y
30 IF Y = -1 THEN 90
40 PRINT Y^2
50 GOTO 20
60 DATA 5, 10, 15, 20, -1
90 PRINT "FINISHED"
99 END
```

(This program contains a loop. Read about loops in II.17.)

There are some limitations on the values you may use in a DATA statement. First, such a value must be a literal or constant--not a variable, and not an expression. The value must be a number or a string; if it is a string, it must be enclosed in quotation marks. Second, any value given in a DATA statement must be of the same type as the variable to which it will be assigned. Note that in line 80 above (on the previous page), the numeric value 60 corresponds to the numeric variable A, and the string value "DOG" corresponds to the string variable B$. BASIC will give an execution error if, at the time the READ is executed, the variable and the value are of different types.

You may use as many DATA statements as you like in a program. The values given in the statements will be "used" sequentially, as required by the execution of READ statements. DATA statements can

(continued)

(II.15)

appear anywhere in the program before the END, and it is a good idea to locate your DATA in a place that makes sense to you. For example, if a section of a program requires READing values from the DATA, put the DATA statements at the end of that section so that you can easily see where the DATA values will be used.

The REOPEN statement moves the "pointer" back to the first value in the DATA list. The next READ statement will then take the first DATA value in the lowest-numbered DATA statement in the program. REOPEN is useful in situations where you want to use the same DATA values, in the same order, more than once.

See Input / Output (II.5), Data Types (II.7), and Variables (II.9).

II.16 DIM

II.16

Use: To establish the size of an array (a subscripted variable).
DIM is short for DIMENSION.

Examples:

```
10 DIM L(15)

10 DIM A$ (50)
```

Remarks:

BASIC needs to know how long an array will be before you refer to any elements or "slots" in the array (for example, before you assign any values to elements of the array). The DIM statement establishes the maximum length. The DIM statement must precede (i.e., have a lower line number than) any statement that refers to an element of the array. Usually, the DIM goes at the very beginning of the program. There must be one DIM statement for every array used in the program.

Only one DIM may be executed for a given array. In the example shown below, line 20 is executed only once each time you RUN the program. BIP will stop execution and print an error message if two DIMs are executed for the same array, or if one DIM for a given array variable is executed twice. This means that you should locate all DIMs outside any loops in your program, so that BASIC executes each different DIM only once.

(continued)

(II.14)

Suppose your DIM statement is
10 DIM X (25)
This means that you may not use more than 25 elements in the array X. Using fewer than 25 will not cause any problems.

This is a simple program using an array. It asks the user for three words, and assigns each word to an element of the array. Then it prints the words in the opposite order.

```
10 DIM LS(3)
20 PRINT "TYPE THREE WORDS, ONE AT A TIME."
30 INPUT LS(1), LS(2), LS(3)
40 PRINT "HERE'S YOUR LIST IN THE OPPOSITE ORDER."
50 PRINT LS(3)
60 PRINT LS(2)
70 PRINT LS(1)
99 END
```

The word "index" is used in connection with arrays to mean the number that specifies each element in the array. (The word "subscript" is also used.) For example, in line 50 above, the index or subscript is the number 3, and it specifies the third element in the array L. "Index" is also used in connection with loops (see II.17) to mean the variable that counts the number of executions of the loop. This program is like the previous example, except that it allows the user to say how long his list will be, and then uses a variable as the index, both of the loop and of the array. It also uses a variable in the -DIM- statement, after that variable has been assigned by -INPUT-.

```
10 PRINT "HOW LONG IS YOUR LIST?"
20 INPUT N
30 DIM LS (N)
40 PRINT "TYPE YOUR WORDS."
50 FOR I = 1 TO N
60 INPUT LS (I)
70 NEXT I
80 PRINT "HERE'S YOUR LIST IN THE OPPOSITE ORDER."
90 FOR I = N TO 1 STEP -1
100 PRINT LS (I)
110 NEXT I
999 END
```

See Primaries (II.8), FOR . . NEXT (II.21).

11.17 Program Flow

When the computer executes a stored program, it follows a predictable path through the list of instructions that is the program. In some programming languages, the order of instructions executed depends simply on the order in which the computer encounters them from the input device (e.g., card by card from a card reader or line by line from a disk file). Other languages (including BASIC, as you know) use line numbers, and the computer executes instructions in numeric order.

In either case, all languages have the ability to tell the computer to follow a different order, to go to a different place in the list of instructions and carry on from there. This is called "branching" and it can be either unconditional or conditional. Unconditional branching refers to a change in the sequence of execution that will always be carried out regardless of anything else in the program. Unconditional branching is something like telling the computer, "Don't ask any questions, just go to a different part of the program." Conditional branching asks a question *first*; whether or not the change in sequence is carried out depends on some condition being true. Frequently it involves looking at a certain variable, and executing the branch if the variable has a certain value. The program specifies a decision to be made by the computer.

The ability to make appropriate decisions constitutes the "smartness" of a program. Virtually no useful program runs straight through all its statements, without ever changing the order of execution.

11.17.1 Loops

A loop is a series of statements that is executed more than once. It is an extremely useful programming structure. By using a loop, you can make the computer do the same thing many times, but you give a set of instructions only once. The general form of a loop is this:

```
Start the loop here.
Have the program do something.
Decide if the "something" should be done again.
If so, go back up and start the loop again.
If not, continue on from here.
```

The "something" can be very complex. It can be most of the program; for example, a program that plays a game can start itself again depending on what information the user gives after playing once--the whole game is inside the loop.

(continued)

A large category of loops follows this general pattern:

Set a "start" value.
Set an "end" value.
Set a counter equal to the start value 1.
Increment the counter.
Do the work.
Look at the end value--if the counter is less than or equal to the end, go back to the "increment" place and continue from there.
Otherwise (i.e., the counter is greater than the end value), continue from here.

A "counter" is a numeric variable that you use to count something. In this case, it counts the number of times the loop has been executed--you increment the counter (add 1 to it) each time you go through the loop. The counter is also called the "index."

This pattern is used in situations where the problem can be solved by performing the same sequence of steps, perhaps with some variations, a number of times. This is "the work." The number of times "the work" is done depends on the "start" and "end" values. For example, the following is a general program (in no programming language) that counts from 1 to 3:

```
Start = 1
End = 3
Counter = start - 1
* Counter = counter + 1          } These three lines
Print value of counter           } are the loop.  The
If counter less than end, go to * } work is to print the
                                   value of the counter.
Print goodbye
```

Different problems require different variations on this general pattern. For example, the "work" may involve a more complicated set of operations, or the counter may be changed by some value other than 1, or the order in which the pattern parts are executed may need to be different. Once the general pattern is understood, however, it is easier to see which details must be changed to solve a particular problem. Look (on the next page) at a program (in no programming language) that counts backwards from a number input by the user. Notice the ways in which it is different from the last example.

(continued)

```
Print hello user, type in a number please
Input a character number the user types
Set n equal
Counter = start
# Print value of counter
Counter = counter minus one
If counter greater than or equal to end, go to #
Print goodbye
```

Look at each other then counter of course. This time the program is going to be interpreting (because) which the user's type or does close so to it the characters. This program doesn't need a start or end value, because it can't control anything, but it does need to make a comparison to decide whether or not to go through the loop again. If she wants two string variables, one to hold the user's name, and one to hold the user's answer to the question yes/no.

```
Print hello user, please type your name
Username = character string <-- user types
# Print shall I say your name? yes or no, please
Answer = character the user types
If answer is no, then go to goodbye line
Print value of username
Go to #
Print goodbye
```

This loop uses both a conditional branch ("If the answer is no, ...") and an unconditional branch ("go to #"). Remember to label areas to use the conditional branch at the top of the loop ("#"), before you "the word"), and then unconditionally go back up and start again once you have checked the answer, as in this example.

It is not hard to write a program that makes the computer do the same thing over and over, again indefinitely. To force an "infinite loop" like this is a hell, hold down the "CTRL" key while you type the letter C. There is a program in Section 11.10 that has an infinite loop.

See 11.10 - 11.21 for the BASIC statements used to construct loops.

Frequently, the same set of instructions is used in many different parts of a program. An efficient way to use these instructions is to set them up in one part of the program and to branch to that part from other parts. The sequence of statements that is accessed from different parts of the program is known as a subroutine.

Since a subroutine can be "called" from different places, it is important for the computer to know where to "return" to after the statements in the subroutine have been executed. Most languages have the ability to remember the place from which execution jumped to the subroutine and then to go back to that place to continue after the subroutine.

For example, consider a program that simulates a game of blackjack. It might include a subroutine that "deals the cards" by generating random numbers and translating those numbers into cards from the deck. In blackjack, the dealer deals cards in two different situations: either it is the beginning of a new hand, or one of the players is asking for another card, in addition to those he holds already. So, in the blackjack program, the card-dealing subroutine would be branched to (or "called") in those different situations. What happens afterwards depends on what was happening when the dealer dealt a card. The branch-and-return capability allows the program to go back to that place after the cards have been dealt, so that play can continue appropriately. In the first case, the program would only check to see if all the cards needed to start the game had been dealt. In the second case, it would have to ask the next player if he wanted another card.

See 11.22 for the BASIC statements used to set up subroutines.

Use: To alter the sequence of execution of the program unconditionally.

Example:

```
70 GOTO 10
```

Remarks:

BASIC executes a program in the order of the line numbers. When you say RUN, it finds the lowest-numbered line and executes that statement. Then it finds the next higher line number and executes the statement on that line. And so it goes---it's very simple. The above example would change that order by sending BASIC back to line 10 every time line 70 was executed.

This program will repeat itself endlessly (until the user types ^C to stop it), counting from 1 on up.

```
10 X = 1
20 PRINT X
30 X = X+1
40 GOTO 20
50 END
```

(To stop a program like this, hold down the "CTRL" key while you type the letter C. See Section I.6.)

Note that once BASIC has executed the line specified in the GOTO statement, it continues execution from that point. In this example, the order of lines executed would be

```
10,
20, 30, 40,          (here GOTO changes things)
20, 30, 40,          (GOTO 20 again)
20, 30, 40,          (and again)
etc.
```

The only way to make this program stop is to hold down the "CTRL" key while you type the letter C. This is called "interrupting" or "aborting" the execution of the program.

If your GOTO statement specifies a non-existent line, BIP will print an error message before it allows you to RUN the program.

See Program Flow (II.17).

The BASIC relational operators are

=	equal to
<>	not equal to
<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to

Relational operators are used to compare two values. This comparison is called a Boolean expression, and its value is always either true or false.

In numeric expressions, the relational operators work as one normally expects them to. In string expressions, relational operators compare the strings for alphabetic order. Thus:

6 = 6	is true
8.7 >= 5	is true
4 <> 8/2	is false
"DOG" > "CAT"	is true
"ALPHABET" < "A"	is false

The Boolean operators are

NOT
AND
OR

Boolean operators are used to combine or change Boolean expressions. Say the variable X has the value 5,
Y has the value 99, and
A$ has the value "YES".
Now consider the following Boolean expressions:

a. NOT

The Boolean expression	X >= 0	is true.
The Boolean expression	NOT X >= 0	is false.
The expression	Y < X*5	is false.
The expression	NOT Y < X*5	is true.

The expression A$ <> "NO" is equivalent to NOT A$ = "NO"

(continued)

(II.19)

b. AND

An expression that includes AND is true only if all its parts are true.

```
X <> 4 AND Y <= 100              is true.
A$ = "NO" AND Y <= 100           is false.
```

c. OR

An expression that includes OR is true if any or all of its parts are true.

```
X <> 4 OR Y > 100                is true.
A$ = "NO" OR X < 5               is false.
Y <> 98 OR A$ = "YES"            is true.
```

Unless parentheses are used, BASIC applies the Boolean operators in this order: NOT, AND, OR. Thus,

```
    NOT A$ = "YES" AND Y < 100
```

is equivalent to

```
    (NOT A$ = YES") AND (Y < 100)
```

and the expression is false, because NOT A$ = "YES" is false.

```
    A$ = "NO" OR NOT X = 6 AND Y > 50
```

is equivalent to

```
    (A$ = "NO") OR ((NOT X = 6) AND (Y > 50))
       (f)           (    (t)         (t) )
```

and the expression is true.

If you want to force BASIC to evaluate your Boolean expressions in a different order, use parentheses as you would with numeric expressions. For example,

```
    NOT A$ = "YES" OR Y < 100
```

is equivalent to

```
    (NOT A$ = "YES") OR (Y < 100)
          (f)             (t)
```

and the expression is true, because Y < 100 is true.

However,

```
    NOT (A$ = "YES" OR Y < 100)
         (t)           (t)
```

is false, because (A$ = "YES" OR Y < 100) is true. Parentheses can make a difference if you need to use complicated Boolean expressions.

See Data Types and Values (II.7).

Use: To modify the order of execution so that your program can do different things in different situations.

Examples:

```
50 IF B > 5 THEN 150

50 IF X$ = "OXYGEN" THEN 300

50 IF A$ = "REPEAT" AND C > 0 THEN 10
```

Remarks:

The IF . . THEN statement is executed in the following way:

a. The Boolean expression following IF is evaluated as either true or false, depending on the values and the relationship within the expression.
b. If the Boolean expression is false, the sequence of execution does not change, and the next line executed will be the line after the line containing the IF . . THEN.
c. If the Boolean expression is true, the next line executed will be that specified by the line number after THEN. (One may say that "control is transferred" to that different point in the program, since execution will continue from that specified line, not from the line following the IF . . THEN statement.)

This short program uses an IF . . THEN to decide whether or not to start itself over:

```
10 PRINT "TYPE YOUR NAME."
20 INPUT N$
30 PRINT "HELLO, "; N$
40 PRINT "TO START OVER, TYPE 'YES'."
50 INPUT A$
60 IF A$ = "YES" THEN 10
70 PRINT "OK.  GOODBYE."
999 END
```

Note that only the word YES from the user causes the program to continue execution (again) from line 10. Anything the user types that is not YES will be taken as a NO answer. This program is another example of a loop. The number of times that the loop will be executed depends entirely on what the user types when the program is run. Try this: Copy this program, then RUN it. Use TRACE or FLOW to see how things work.

See Program Flow (II.17) and Boolean Expressions (II.19).

Use: To have BASIC do the counting, incrementing, and checking in a loop, automatically.

Examples:

```
    10 REM SQUARES FROM 1 TO 5     see II.24 about REM
    20 FOR N = 1 TO 5              establish "start" and "end"
    30 PRINT N                     do something
    40 PRINT N^2                   do something else
    50 NEXT N                      add 1 to N.  If N is 5 or less,
                                     go to 30 again.  If N is
                                     more than 5, continue to 99.
    99 END
```

```
    10 REM COUNT FROM 10 TO 1      !! counts backwards
    20 FOR N = 10 TO 1 STEP -1      because the step is negative
    30 PRINT N
    40 NEXT N
    99 END
```

Remarks:

FOR. .NEXT loops save the programmer some work by automatically incrementing the counter and checking its value against the top value. The general form of the FOR statement is

FOR <index> = <start> TO <end> STEP <howmany>

FOR. .NEXT loops are executed in this way:

a. The "index" variable is assigned the value of <start>.
b. The statements following the FOR statement are executed in order.
c. When the NEXT statement is encountered,
    1. The value of <howmany> is added to the index. If no STEP is included, 1 is added. (The value of the index moves closer to <end>.)
    2. If the value of the index has not passed the <end> value, the statements following the FOR statement are executed again --- the loop is repeated with the new value of the index.
    3. If the value of the index has passed the <end>, the loop is not repeated, and execution continues from the statement after the NEXT statement.

(continued)

The FOR statement sets up the "start" and "end" values for the loop, and marks its beginning. The NEXT statement marks the end of the loop. The value of the index variable (N in the examples above) is changed, and checked against the "TO" value, when the NEXT statement is executed. All the "work" lies between the FOR and the NEXT.

The following three programs illustrate how loops work. All three programs do the same thing: they all count by twos from two to twenty. The first program is pretty silly, since it makes the programmer do more work than is necessary:

```
10 PRINT "COUNTING BY TWOS"
20 PRINT 2
30 PRINT 4
40 PRINT 6
50 PRINT 8
60 PRINT 10
70 PRINT 12
80 PRINT 14
90 PRINT 16
100 PRINT 18
110 PRINT 20
120 PRINT "WHEW"
999 END
```

The second program is much better, since it makes the computer do more of the work:

```
10 PRINT "COUNTING BY TWOS"
20 N = 2
30 PRINT N
40 N = N + 2
50 IF N <= 20 THEN 30
60 PRINT "FINISHED"
99 END
```

The third program is even better, since it takes advantage of the automatic features of the FOR . . NEXT structure:

```
10 PRINT "COUNTING BY TWOS"
20 FOR N = 2 TO 20 STEP 2
30 PRINT N
40 NEXT N
50 PRINT "THAT'S ALL, FOLKS!"
99 END
```

(continued)

(II.21)

It is sometimes very useful to put one loop inside another, or "nest" the two loops. The following program might be used by the principal of a school to add up the number of students in each grade and in the school as a whole. The "outer loop" is indexed by the variable I, and the "inner loop" is indexed by J. The extra lines on the left show you how the J-loop is nested inside the I-loop.

```
      10 T = 0
      20 S = 0
      25 REM T IS FOR TOTAL IN THE SCHOOL, S IS FOR GRADE SUBTOTALS
      30 PRINT "HOW MANY GRADES DO YOU HAVE IN THIS SCHOOL?"
      40 INPUT G
 ---- 50 FOR I = 1 TO G
 |    60 PRINT "HOW MANY CLASSROOMS DO YOU HAVE IN GRADE "; I
 |    70 INPUT C
 |  - 80 FOR J = 1 TO C
 |  | 90 PRINT "HOW MANY KIDS IN CLASS "; J; " IN GRADE "; I
 |  | 100 INPUT K
 |  | 110 S = S + K
 |  | 115 REM ADD THOSE KIDS TO SUBTOTAL FOR THE GRADE
 |  - 120 NEXT J
 |    130 PRINT "IN GRADE "; I; " YOU HAVE "; S; " STUDENTS"
 |    140 T = T + S
 |    145 REM ADD TOTAL FOR THIS GRADE INTO THE TOTAL FOR THE SCHOOL
 |    150 S = 0
 |    155 REM SET THE SUBTOTAL BACK TO ZERO, READY FOR NEXT GRADE
 ---- 160 NEXT I
      170 PRINT "IN THE WHOLE SCHOOL YOU HAVE "; T; " STUDENTS"
      999 END
```

One thing to remember when you nest loops is that the inner loop(s) must be entirely contained inside the outer loop. BIP won't let you RUN the program if it has loops like this:

```
10 FOR X = 1 TO 10
      .
      .
40 FOR Y = 10 TO 100 STEP 10
      .
      .
70 NEXT X
      .
      .
90 NEXT Y
```

The NEXT for the Y-loop is outside the X-loop completely, which is not allowed.

See Program Flow (II.17).

Use: To transfer execution to a subroutine, then to return back to the same place.

Remarks:

A sequence of statements that is accessed from different parts of the program is called a subroutine. BIP subroutines are somewhat different from subroutines in other implementations of BASIC. A BIP subroutine is a sequence of statements that come between a BEGINSUB and an ENDSUB. The sequence is only "called" by a GOSUB. It can terminate either with a RETURN or the ENDSUB, both of which cause a jump back to the line after the GOSUB that called the subroutine.

Subroutines are useful in a program that uses the same sequence of statements in a number of different situations, in that they allow the programmer to write the sequence only once and yet have it accessible from many different parts of the program. When this sequence has been executed, control returns to the place from which the sequence was called. Complicated programs are also much easier to debug if they have subroutines corresponding to the different parts of the job the program is intended to do. See "Branch and Return" in Section II.17.2.

Example:

```
.
.  (other lines of the program)
.
50 GOSUB 800
60 PRINT "WE RETURN FROM THE SUBROUTINE."
70 GOTO 999
.
.
800 BEGINSUB "NUMERO UNO"
810 INPUT X
820 IF X = 1 THEN 850
830 PRINT "X IS NOT 1.  YOU LOSE."
840 RETURN
850 PRINT "X IS 1.  YOU GET A STAR."
860 PRINT "* * * * *"
870 ENDSUB "NUMERO UNO"
999 END
```

When line 50 is executed, control is transferred to line 800. Execution continues with 800, 810, and 820. If X equals 1, the next lines executed are 850, 860, 870, and then back to 60. If X is not equal to 1 at line 820, the sequence is 830, 840, and then back to 60.

(continued)

Notice these requirements of each of the four statements:

GOSUB 50 GOSUB 800
jumps into the subroutine.
Line 800 must be a BEGINSUB.

BEGINSUB 800 BEGINSUB "NUMERO UNO"
beginning of the subroutine. The name (whatever you like, enclosed in quotes) is optional and has no effect except to help you see what your program is doing.

RETURN 840 RETURN
jumps to the line following the GOSUB; in this case, line 60. Use as many RETURNs as you like, for conditional branching out of the subroutine.

ENDSUB 870 ENDSUB "NUMERO UNO"
marks the end of the subroutine. It causes an automatic RETURN to (in this case) line 60. The name is optional — use it to match up with the BEGINSUB name if it helps you.

Notice that a BIP subroutine must begin with a BEGINSUB and end with an ENDSUB, and that these statements must be accessed only by the GOSUB. A BIP subroutine does not require you to use a RETURN, since ENDSUB includes its function. In BIP, RETURN and ENDSUB are similar to STOP and END: you may use as many RETURNs and STOPs as you need (including none at all), but you must use one END per program and one ENDSUB per subroutine.

There are no jumps into a subroutine except by a GOSUB to its BEGINSUB, and no jumps out of a subroutine except by a GOSUB (to another subroutine), a RETURN, or an ENDSUB. Look at these pairs of programs for illustrations of the syntax of subroutines:

*** no jumping in ***

```
This example is illegal        This example is legal
10 INPUT X                     10 INPUT X
20 IF X = 1 THEN 100           20 IF X <> 1 THEN 40
.                              30 GOSUB 100
.                              40 STOP
.                              .
.                              .
100 BEGINSUB                   100 BEGINSUB
.                              .
.                              .
```

(continued)

*** no "flow through" into the subroutine ***

```
Illegal                     Legal
10 GOSUB 100                10 GOSUB 100
20 PRINT "X"                20 PRINT "X"
.                           30 STOP
.                           .
.                           .
100 RETURN                  100 RETURN
```

(the problem with the illegal example is that without the STOP statement, BASIC could reach and execute the RETURN directly in the sequence of line numbers, which is illegal. A RETURN may only be executed immediately after its matching GOSUB.)

*** no jumping out ***

```
Illegal                     Legal
10 GOSUB 100                10 GOSUB 100
20 STOP                     20 STOP
.                           .
.                           .
100 RETURN                  100 RETURN
110 INPUT X                 110 INPUT X
120 IF X = 1 THEN 20        120 IF X = 1 THEN 140
130 PRINT "X IS NOT 1"      130 PRINT "X IS NOT 1"
140 ENDSUB                  140 ENDSUB
```

*** no subroutine calling itself ***

```
Illegal                     There is no right way for this.
10 GOSUB 100                BASIC is not recursive.
.
.
.
100 REMARK
110 PRINT "IN THE SUBROUTINE"
120 GOSUB 100
130 ENDSUB
```

See Program Flow (II.17)

Imagine this exchange. You say, "Double nine number; 6" and your friend says, "Okay: 12." To double a number is to use that number in a specific way and then to give the result back. In this example, "double" is a function, the number 6 is the argument to the function, and the number 12 (the result of doubling 6) is the value returned by the function.

A function is some defined process that produces a result. It may require no arguments, like the function that picks a random number (see RND). It may require one argument, like the function that doubles a number--you can't double something without knowing what that something is. Or it may require more than one argument, like the function that finds the smaller of two numbers--you can't say something about two numbers without knowing what they both are.

A function always returns one value.

Keep the special meanings of argument and return in mind. Don't confuse them with the regular English meanings of the words.

You may think of a function as a shorthand for some series of operations. The value returned by a function is used like any other value in the programming language you are using: you may assign it to a variable, or use it in a Boolean expression, or print it, etc. Some examples of functions are given in the next few pages.

To generate a random number is simply to tell the computer to pick a number. One of the most interesting uses for random numbers is in programs that play games: dealing cards, choosing a number for the user to guess, or choosing a move in tic-tac-toe, for example.

11.23.1 Built-in Functions 11.23.1

BASIC has several built-in functions. That is, there are some operations that are so frequently used by programmers that they have been added to the commands that the interpreter understands. The exact list of these functions will vary with the implementation of BASIC, and the list is sometimes called a "library". The following functions are built into DEC's BASIC:

(continued)

Use: To generate a random number

Examples:
```
20 X = RND

20 PRINT RND*10

20 X = INT (RND * 10 + 1)
```

Remarks:
The RND function returns a random number greater than 0 and less than 1. That is, it makes the computer "pick a number" at random the way you might pick a card from a deck. RND always picks a decimal fraction between 0 and 1, so read about INT for interesting ways to generate and use random integers.

Use: To convert a real number into an integer.

Examples:
```
30 X = INT(7.6)

30 PRINT INT (-67.89)

30 X = INT (RND * 10 + 1)
```

Remarks:
BASIC thinks of all numbers as real numbers (i.e., as numbers with decimal fractions), not as integers. There are many situations in which a program should work with only the "integer part" of a number, and the INT function does the job.

But's BASIC, unlike some other implementations, interprets INT to mean "return the largest integer that is not greater than the argument." This means that:
```
INT (7.6)  = 7
INT (-7.6) = -8
```
because -8 is the largest integer that is not greater than -7.6.

The argument to the INT function must evaluate to a number. INT(5*3) is legal, but INT(A$) is not, because A$ cannot be a number.

(continued)

Some uses of INT include:

a. Generating random integers (see RND).
The INT function returns a random number between 0 and (n-1) (where n=the number function). To create an integer, you must first multiply the random number by 10 (or integer must be in base 1) and then convert it to an integer:

    INT (RND*10)

RND() returns a random integer between 0 and 9, inclusive. The value of (RND*10) will be greater than 9 and less than 10. RND() cannot equal 0 the 10 (0.0) as a right of 0.99.

    INT (RND*n + 1)

RND() returns a ...random integer between 1 and 10, since the range of values (because n=0 is applied) is 1.0 to 10.99. Once Basic generates another random value in the variable N:

    N = INT (RND*10 + 1)

In general,

    INT (RND * N + A)

RND() returns a random integer between A and N inclusive.

b. Dividing "evenly".
If a number N divides another number A evenly, then A/N is an integer which is termed "divisible by". The Boolean expression

    A/N = INT (A/N)

will be true only if N is evenly divisible by A. For example, the Boolean expression

    12/4 = INT(12/4)

is false, because 12/4 equals 3.00, and INT(3.00) equals 3.

But (6/3) = INT(6/3)
is true, because (6/3 equals 2, and INT(2) equals 2.

This property uses INT to determine if one integer number given is evenly divisible by the second number:

```
10 PRINT "GIVE TWO DIVISORS"
20 INPUT A
30 PRINT "GIVE TWO DIVIDEND"
40 INPUT B
50 IF A/B = INT(A/B) THEN GO
60 PRINT "BUT GIVEN YOU AGAIN."
70 GOTO 10
80 PRINT A; " IS EVENLY DIVISIBLE BY "; B
90 END
```

Use: To return the square root of a numeric expression.

Examples:

```
30 S = SQR(25)

30 IF SQR (7*10) > R THEN 10

30 PRINT "THE SQUARE ROOT OF R IS "; SQR(R)
```

Remarks:

The SQR function finds the positive square root of its argument. The only restrictions on the argument are:

a. It must be an expression that evaluates as a number.
b. It must be greater than or equal to zero, since negative numbers do not have real square roots.

Use: To return the length of a string.

Examples:

```
30 INPUT T$
40 L = LEN (T$)

30 READ C$
40 X = LEN (C$)
```

Remarks:

The LEN function counts the number of characters in its string argument. If the value of T$ was "TOMATO", the function would return the value 6.

Use: to return the value of any expression the programmer wants to use often.

Examples:

```
30 TWICE (N) = N*2
40 IF TWICE (I) > 100 THEN 10
50 REM BACK TO 10 IF I TIMES 2 IS BIG


30 CONCAT (R$) = R$ & R$
40 INPUT D$
50 PRINT "I'LL REPEAT AFTER YOU - "; CONCAT (D$)
```

Remarks:

Most implementations of BASIC, including BIP, allow you to define your own functions. In BIP, functions may have only one argument. Both string and numeric functions may be defined. For example,

```
10 ADDER (X) = X+1
```

defines a numeric function named ADDER, whose argument is X, and whose value is X + 1.

Defining a function to do something that you have to do more than once saves you some trouble in writing your program. For example, if your program had to generate lots of random numbers (see RND and INT, above), you might define that function, then just call it each time you needed a random number. This program is a simplified illustration:

```
10 PICKME (X) = INT (RND * X + 1)
20 REM "PICKME" WILL PICK AN INTEGER BETWEEN 1 AND X
30 PRINT "HERE'S A NUMBER BETWEEN 1 AND 10:"
40 PRINT PICKME (10)
50 PRINT "AND HERE'S A NUMBER BETWEEN 1 AND 5:"
60 PRINT PICKME (5)
99 END
```

You might copy and run this program a few times to see how all these functions work together.

You may define a given function only once in a program, but you may use as many different functions as you like. The kind of expression used in a function must match the data type of the argument: If the argument is a numeric variable, the expression must be numeric, and if the argument is a string variable, the expression must evaluate as a string. The name of the function must be at least 3 letters long. It can be very long (20 letters), but since the purpose of functions is to save on typing, your function names should probably be less than 10 letters long. You may not use "special characters" like periods, commas, or semi-colons in the function name.

II.24 Other Useful Statements II.24

II.24.1 STOP II.24.1

Use: To tell the computer that it has finished executing your program.

Example:

```
50 STOP
```

Remarks:

Every BASIC program must have an END statement. The END statement must have the highest line number in the program.

In addition, you may use as many STOP statements as you like. STOP is equivalent to END, except that STOP may have any line number. STOP statements are useful in programs that may terminate in many ways.

BIP's BASIC always prints the number of the last line executed when a program terminates. Using STOP statements can be very valuable in debugging a program that has many parts--it can help you locate problems by causing execution to terminate under certain conditions without confusing the issue by continuing execution with wrong values. Then the line number at which the program terminated can help you see what erroneous condition occurred.

See END (II.4) and GOSUB (II.22).

II.24.2 REM II.24.2

Use: To write REMarks inside your program, making it easier to understand.

Examples:

```
60 REM !!! STOP LOOPING IF X IS TOO BIG.

200 REM THE FOLLOWING 5 LINES CALCULATE THE AVERAGE:
```

Remarks:

Use a REM statement whenever you like. It does not affect the execution of your program in any way, but it gives you a way to make notes about the program as you go along, inside the program itself. You may also use a REM statement with a blank line just to make a break between blocks of lines in your program.

SECTION III. BIP COMMANDS

Whenever you deal with BASIC, you are really communicating with the computer on two levels. One level connects you with the BASIC language and the computer's ability to execute programs written in BASIC. The other level connects you with a more general operating system, which allows you some control over the world in which your own programs live. In this course, the general system is BIP, the program that runs everything you see happening at your terminal. Through BIP, you can write and execute programs in BASIC; in addition, you are presented with programming tasks and you are allowed to save and modify your programs. Some of the commands in this section are identical to those in other implementations of BASIC and some are peculiar to BIP. You will just have to learn other commands when you use other versions of BASIC.

III.1 Curriculum Manipulation

These commands deal with the programming tasks that form the instructional base of BIP.

TASK	Start a new problem. BIP will select it for you.
HINT	Print a hint. Some tasks have no hints; some have more than one. Type HINT to help you understand what the task requires. For another kind of hint, see REP below.
SUB	Present a sub task. Some tasks are broken down into parts. If you can handle all the parts, you can write the program for the task. Ask for SUBs whenever you have trouble getting started on a complicated task. Finish a sub task with MORE or ENOUGH.
MORE	Continue the current problem. BIP does some checking of your program before allowing you to continue. MORE in a sub task returns you to the task you came from.
ENOUGH	End the current task immediately. BIP does not check your program, and keeps no record of your having entered that task. ENOUGH in a sub task returns you to the task you came from.

(continued)

MODEL — Print out a model solution to the current task. The model solution is not necessarily the only way to write the program. BIP does not take you out of the task.

DEMO — Execute the model solution. The demo should help you write your own program by demonstrating one possible solution to the task.

DEMO TRACE — Execute the model solution and show what's happening at the same time. BIP prints the number of each line of the model solution as it is executed and prints the value of each variable each time it is assigned. Once you have run the DEMO a few times, you know what the model solution does. Then the DEMO TRACE will help you see how the model works. See TRACE in Section III.2. If the screen is flashing by too fast, use the HOLD key. (See Section I.6.)

REP — Present a flowchart-like representation of the model solution. REP has its own set of commands that allow you to expand the representation to find out what the structure of the program looks like.

When in REP, look at the top of the screen for a reminder of what you can do. You can always start with A<cr>. (Remember, <cr> is carriage return.) Watch the screen. Then type B, C, D, etc., or A1, A2, B1, B2, etc. (followed always by <cr>) if these letters and numbers appear in boxes on the screen or at the top.

You can start over by typing RESET, or get out by typing EXIT.

^ and / will move the screen up and down for you when the flowchart gets too big to see all of it at once. (Type shift-N to get the ^ "up-arrow" character.)

You cannot write your program in REP. Type EXIT<cr>, wait for the * at the left margin, then write your program or do whatever else.

Type a "?" to get help from REP.

These commands do not deal with the curriculum, only with the program you are currently writing and running.

LIST — Print out the current program. Use this to see what your entire program looks like---it helps.

EDIT <line number>
Change a line without typing out the whole thing all over again. See Section I.6 for the control characters that let you change the line. (Editing is much easier to demonstrate than to explain. Try following the example in I.6.)

SCR — Delete ("scratch") the current program, wiping the slate clean so you can start afresh.

RUN — Execute the program---have BASIC follow your list of instructions. To stop execution of your program before it reaches its END, interrupt like this: hold down the "CTRL" key while you type the letter G. See Section I.6.

SEQ <starting> <increment>
Renumber the lines of the program. <starting> is the first line you want to have "reSEQuenced," and <increment> is the distance you want to have between the lines. For example,

```
SEQ 100 20
```

will renumber the lines in your program from line 100 upward, and each new line number will be 20 more than the line number that precedes it. (The new numbers in this example, starting at 100, would be 100, 120, 140, etc.) Use SEQ when you want to reorganize your program to make more space available between the existing lines, so that you can insert new lines into the program.

SEQ also changes the line numbers specified in GOTO, IF. .THEN, and GOSUB statements so that the program executes exactly as it did before you decided to reSEQuence the line numbers.

TRACE — Execute the program and show what's happening at the same time. BIP prints the number of each line as it is executed, and prints the value of each variable each time it is assigned. This is an extremely valuable debugging tool. Use it on a simple program first, to see exactly what it does. Then use it any time your program does not seem to do what you intended.

(continued)

TRACE <number1>

Executes the whole program. The trace will start as soon as the line numbered <number1> is executed, and the trace continues to the end of the program. Use this command if you know that the first part of your program is correct and you want to avoid taking the time to trace through things that already work.

TRACE <number1> <number2>

Executes the whole program. In addition, it TRACEs execution of all lines whose numbers are between <number1> and <number2>.
For example,

```
TRACE 100 200
```

executes the entire program, and prints line numbers and variable values between lines 100 and 200 inclusive.

Example of TRACE:

For the program:

```
10 FOR J = 1 TO 2
20 LET X = J
30 NEXT J
40 PRINT "FINISHED!"
99 END
```

Typing "TRACE" will produce this output:

```
TRACE STARTING AT LINE 10
10:    J = 1
20:    X = 1
30:    J = 2
20:    X = 2
30:    J = 3
40: FINISHED!

99:
EXECUTION COMPLETED AT LINE 99
```

FLOW

FLOW is a debugging aid that allows you to watch your program execute. Your program is displayed on the screen, and the number of the current line being executed blinks. To execute the next line, press <CR>.

By stepping through the program in this way, you have time to examine the program and the sequence of execution.

(continued)

(III.2)

If a backward jump or a forward jump of more than one statement is executed, an arrow is drawn from the statement currently executing to the statement to be executed next.

The listing of your program does not include the body of any subroutines. When a GOSUB statement is executed, the program on the screen is replaced by the subroutine referenced by the GOSUB instruction. In the upper right hand corner, the display indicates "MAIN PROGRAM" or "LEVEL XX", where XX is the number of GOSUB's that have been executed before a RETURN or an ENDSUB is executed.

Any input or output from the program is displayed on the bottom three lines of the screen. The I/O lines only show the result of the three most recent I/O statements, as older results disappear from the screen.

To stop the execution of your program before it reaches its END, interrupt like this: hold down the "CTRL" key while you type the letter G. See Section I.6.

FLOW <number>

Automatically executes up to the line given without waiting for your <cr> (unless it executes an INPUT statement, which always waits for you to type a value). When the specified line is reached, FLOW returns to its normal mode, and waits for a <cr> after each line is executed. This feature is useful for FLOWing through a long program that is largely correct.

FLOW <variable>, <variable>, . . .

At the top of the screen, there is space for up to SIX trace variables, selected by you. The current value of these variables is displayed and updated whenever a traced variable changes its value. You may trace fewer than six variables. To trace a subscripted variable, type its name without any subscripts.

Sample FLOW commands, explained:

FLOW Executes the whole program, waiting for your <cr> at each line. Does not trace any variables.

FLOW X, Y

Executes the entire program, waiting for your <cr> at each line. Shows the values of the variables X and Y at the top of the screen.

(continued)

FLOW 200, A$, L

Executes automatically up to line 200, displaying the values of A$ and L as it goes. Stops and waits for you to type <cr> when it reaches line 200, continuing in this step-by-step mode through the end of the program.

Notice that you may give either the line number specification, or the variables-to-be-traced specification, or both, or neither. If the line number option is used, it must come first. The order in which you type the variable names, if you use that option, only determines the order in which they are displayed. It does not affect the execution of the program in any way.

These commands allow you to keep your programs for later use. If you do not save a program, it will disappear when you sign off. When you save a program, you must give it a name. The name can be anything you like, but it should not contain any "special characters" like periods, commas, or semi-colons. Once the program has been saved, it is called a "file."

FILES	List the names and dates of all files currently saved in permanent storage. The date and time shown tell you when the file was last SAVEd. The length is the number of lines in the SAVEd program.
SAVE <name>	Store the current program under the <name> given. The name must not be longer than 30 characters. The program is not affected—it is simply copied to a permanent storage area.
GET <name>	Retrieve the file of the <name> given. The current program is SCRatched and replaced by the <name> file. The permanent storage of <name> is not affected. (See comments below.)
MERGE <name>	Retrieve the <name> file from storage and add it to the current program, without SCRatching the current program. BIP will print the messages DUPLICATED LINE and WAS: . . . if the MERGEd file and the current program have lines with the same line number. The "new" line from the merged file will replace the "old" line that was already part of the current program. See comments below.
KILL <name>	Erase the <name> file from permanent storage. The current program is not affected.

It is a good idea to LIST your current program before you SAVE it, to verify that it is what you want. Always LIST before you GET, to prevent automatically SCRatching the current program. Be careful with KILL, since it is final.

(continued)

(III.3)

Your "current program space" and "permanent storage area" are two separate things that only communicate with each other when you use these commands. Remember that SAVE and GET make copies from the current program to permanent storage and vice versa. When you GET a file, BIP copies the file from permanent storage into your current program space, and leaves the permanent file exactly as it was. If you then make some changes to the program, you must SAVE it again if you want the changes to be permanent.

For example, suppose you have SAVEd a program under the name DOG, and then sign off. The next day you GET DOG and make some changes to it. If you then say SAVE CAT, your permanent storage will have both DOG (the old version) and CAT (the new one). If you say, instead, SAVE DOG, then BIP will say "OLD VERSION DELETED" and you will have only the new version, under the name DOG. The moral is: If you want to have two versions of the program, SAVE the revision with a new name. If you don't need the old version any more, SAVE the new version with the same (old) name. If you don't SAVE it at all, the new version (your current program) will disappear when you sign off, and only the old version will be in permanent storage.

WHO — Print the name and student number of the person using the terminal. Use this if someone has left the terminal without signing off. (If you sign him off, he may lose a program, so try to find him first.)

WHAT — Print the name of the current task you are in. This also allows you to have the problem text printed out for you again, without restarting the task.

WHEN — Print the date and time. Obvious use.

FILE — Leave a message for the people at Stanford. Use this whenever you have a problem that you think Stanford should know about. Please describe the problem as thoroughly as you can. Messages left through FILE are read daily, or thereabouts. Type the ESC key twice to end your message.

CALC — evaluates an expression. The expression can be numeric, string, or Boolean. For example,
CALC 5+4
would make BIP print 10. Or
CALC "DOC" & "FOOD"
would make BIP print DOGFOOD. Or
CALC 5=4
would make BIP print FALSE.

CALC cannot evaluate expressions containing variables.

[illegible]

[illegible]

[illegible] [illegible] [illegible] [

constant	Another word for "literal." See II.8.
control character	A special command to BIP. See I.6.
counter	A numeric variable used to count something: usually incremented every time some condition is satisfied. See II.17.
data	In general, information used by a program. See II.2.
DATA	The BASIC statement that provides values to a READ statement. See II.15.
decisions	BASIC's ability to modify the order of execution of your program, depending on certain conditions. See II.17 - II.20.
DEMO	The BIP command that executes the model, to show you how one solution to the current task works. DEMO TRACE executes the model, and traces the values of all its variables at the same time. See III.1.
DIM	The BASIC statement that specifies the maximum number of elements in an array; usually goes at the beginning of a program using arrays. See II.8, II.16.
EDIT	The BIP command that makes it possible for you to change a line in your program without typing it over. See I.6, III.2.
END	A required BASIC statement which must be the last line in the program, terminates execution. See II.4.
ENDSUB	The BIP BASIC statement that ends a subroutine. See II.22.
ENOUGH	The BIP command that terminates the current task without completing it. See III.1.
error	Something that BASIC knows it cannot handle correctly. BIP prints out an error message to tell you what it knows about the error. See I.7.
evaluation	The process by which BASIC determines the value of an expression. See II.7 - II.8, II.19.

execute — Make the computer do something. BASIC is said to execute the lines of a program, i.e., to follow each instruction in the program. See II.2.

expression — Part of a BASIC statement to be evaluated: a primary or operations on primaries. See II.12, II.19.

FILES — The BIP command that lists the names of the files in permanent storage. See III.3.

FIX — The BIP command that allows you to leave a message for Stanford. See III.4.

FLOW — The BIP command that executes your program visibly, one line at a time. See III.2. To interrupt execution, hold down the "CTRL" key while you type G. See "interrupt" below.

FOR . . NEXT — The pair of BASIC statements that sets up a machine-made loop. See II.17, II.21.

function — A defined process that produces a result, e.g., RND, INT, SQR, LEN. See II.23.

GET — The BIP command that retrieves a previously SAVEd program so that you can work on it again. See III.3.

GOSUB — The BASIC statement that causes a jump to a subroutine. See II.17, II.22.

GOTO — The BASIC statement that allows you to alter the sequence of execution unconditionally. See II.17, II.18. Also see "interrupt" below.

HINT — The BIP command that prints a hint to help you with the current task. See III.1.

IF . . THEN — The BASIC statement that allows you to alter the sequence of execution if some condition is true. See II.17, II.20.

increment — To add to the value of a numeric variable, frequently a variable used as a counter.

index	In an array variable, the number in parentheses that specifies each element in the list. See II.8, II.16. In a loop, the number (counter) that keeps track of the number of times the loop has been executed. See II.17, II.21.
input	The set of values supplied to the program; the information on which it operates. See II.5.
INPUT	The BASIC statement that allows the user to assign a value to a variable during execution. See II.14.
INT	the BASIC function that returns the integer part of a real number. See II.23
interrupt	To stop the execution of a program before it reaches its END. Hold down the "CTRL" key while you type the letter G. See I.6.
KILL	The BIP command that erases a file from permanent storage. See III.3.
LEN	The BASIC function that returns the number of characters in a string. See II.23.
LET	The BASIC statement that assigns a value to a variable. See II.11.
line number	An integer that must precede each BASIC statement; statements are executed in order of increasing line numbers. See II.3.
LIST	The BIP command that prints out your program in the order of the line numbers. See III.2.
literal	A primary whose value is itself (as opposed to a variable). See II.8.
location	The place in the computer's memory where a value can be stored; the place or "box" named by a variable. See II.10.
loop	General term for a series of statements whose execution is repeated. See II.17, II.21 To interrupt, hold down the "CTRL" key while you type G. See "interrupt" above.

MERGE	The BIP command that retrieves a file from permanent storage and adds it to the current program. See III.3.
MODEL	The BIP command that prints a typical solution to the current task. See III.1.
MORE	The BIP command that presents the next part of a task. Type it after completing a program. See III.1.
numeric	Having to do with numbers and their values. See II.8.
operation	The process by which two expressions are used to specify a new value:
numeric:	Addition, subtraction, multiplication, division, exponentiation.
string:	Concatenation, substring.
relational:	An operation that compares two string or numeric expressions in some way to produce a Boolean expression.
Boolean:	An operation that combines two Boolean expressions into a new Boolean expression.
	See II.12, II.19.
operator	The symbol for an operation:
numeric:	+ - * / ^
string:	& (start, stop)
relational:	= <> < > <= >=
Boolean:	NOT AND OR
	See II.12, II.13, II.19.
output	The visible results of a program's execution on the terminal. See II.5.
primary	An expression without any operation--either a literal or a variable. See II.7 - II.8.
PRINT	The BASIC statement that produces visible results by causing the terminal to type something. See II.6.

program	A list of instructions for a computer to follow, written in a language that the computer understands. See II.1.
READ	The BASIC statement that assigns a value to a variable; the value is stored in the program in the DATA statement. See II.15.
REM	The BASIC statement that does nothing. It simply allows the programmer to make notes within the program. See II.24.
REOPEN	The BASIC statement that moves the "Read-data pointer" back to the first DATA value in the program. See II.15.
REP	The BIP command that presents a flowchart-like representation of the model solution to show you its structure. See III.1.
return	To determine and give back a value. All functions return a value. See II.23.
RETURN	The BASIC statement that causes a jump back from a subroutine to the place from which the subroutine was called. See II.22.
RND	The BASIC function that returns a random decimal fraction between 0 and 1. It requires no arguments. See II.23.
RUN	The BIP command that tells the computer to execute your program. See III.2.
SAVE	The BIP command that puts your current program into permanant storage for your next session. See III.3.
SCR	The BIP command that erases your current program. See III.2.
SEQ	The BIP command that renumbers the lines in your program to give you more available space between the existing lines. See III.2.
signing off	Ending a session on the computer. Signing off is achieved with BYE. See I.3.
SQR	The BASIC function that returns the positive square root of its numeric argument. See II.23.

statement	A single BASIC instruction occupying one line of the program. See II.1 - II.3.
string	A group of characters in a particular order. See II.7 - II.8.
STOP	The BASIC statement that may appear at any place in the program and terminates execution of the program. See II.24.
SUB	The BIP command that presents a sub task. Use it to get the current task broken down into parts. See III.1.
subscript	a number or numeric variable in parentheses that specifies an element of an array.
subscripted	A kind of variable, one that can contain more than one value at one time. See "array". See II.8, II.16.
substring	A part of a string. See II.13.
subroutine	A sequence of BASIC statements that can be accessed and executed from different places in the main program, returning back to the place from which it is called. See II.22.
TASK	The BIP command that presents the next programming task. Type it after completing the previous task. See III.1.
TRACE	The BIP command that both executes a program and prints out line numbers and variables as execution progresses. See III.2. To interrupt execution, hold down the "CTRL" key while you type G. See "interrupt" above.
user	In general, the person who runs a program. Frequently, also the person who wrote it.
user-defined function	A function defined in your program, which returns the value of the expression that you specify. See II.23.
value	The result of evaluating an expression or a function. Either a number, or a string, or TRUE or FALSE. See II.7, II.12, II.19

variable	A name for a location in the computer's memory, a "box" that can hold a numeric or string value. See II.8 - II.11.
WHAT	The BIP command that tells you the name of your current task and allows you to see the problem text again. See III.4.
WHEN	The BIP command that tells you the date and time. See III.4.
WHO	The BIP command that tells you who is signed on at the terminal. See III.4.